The Runtime of Algorithms

The typical measure of running time is to count the number of operations performed.

- What is wrong with this measure?

- Suppose we have two algorithms to solve the same problem; that problem has, as input, an array A of size n. Would it be better to have an algorithm that takes $20n$ operations or one that takes n^2 operations? Why?

In general, we are concerned with solving very big problems. As such, our question becomes "as n becomes very large, which is better?"

Based on this, a good question to ask is: "if I double the size of the input, how much longer does the algorithm take?" For each of the following operation counts, how much longer will the algorithm take if we do this?

Time	Change	Time	Change
n		$20n$	
$10n+37$		n^2	
n^4		n^5+10n^3+21	

Linear-time algorithms double in running-time when the size of the input doubles.

Polynomial-time algorithms increase by a constant factor.

Finding the running time of an algorithm

Consider the following algorithm:

`Find-Max`
Input: an array A of n comparable values, denoted $A_1 \ldots A_n$
Output: the value of the largest element of A.

```
max = $A_1$
for $i = 2 \rightarrow n$ do
  if max < $A_i$ then
    max = $A_i$
return max
```

How many lines of code get executed when `Find-Max` is run, as a function of n, the number of elements in A?

Consider the following algorithm:

```
Bubble-Sort
**Input**: an array $A$ of $n$ comparable values, denoted $A_1 \dots A_n$
**Effect**: Array $A$ contains the same elements, in sorted order, when this returns.
  for $i = 1 \to n - 1$ do
    for $j = 1 \to n - i$ do
      if $a_j > a_{j+1}$ then
        swap $a_j$ and $a_{j+1}$
```

How many lines of code get executed when `Bubble-Sort` is run, as a function of n, the number of elements in A?

We will typically give the number of operations it takes in *the worst case*.

- What other ways could we analyze the number of operations?
- What kind of reasons can you think of for why we would want to analyze worst case?
- What kind of problems can you think of for analyzing in other ways?

Notation for growth of functions

We say that $f(n)$ is $O(g(n))$ (read: f of n is big-oh of g of n) if and only if:

For some constants c and n_0, for all $n > n_0$, $f(n) \leq cg(n)$.

Alternatively, $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$

(This is sometimes written as $f(n) = O(g(n))$)

Effectively, we are ignoring constant factors in our analysis. Are these factors unimportant?

- $10n^3$ is $O(?)$
- $20n^2 + 13n + 5$ is $O(?)$
- Is it the case that $10n^3$ is $O(n^4)$?
- $f(n) = \log_{10} n$ and $g(n) = \log_2 n$. How do they relate?
- $f(n) = \log n$. What base do I mean?

O-notation means upper bounds! That is *imperfect* knowledge on *our* parts.

- Is our objection that operations take unequal amounts of time a problem in this model?

- What does this mean? "Algorithm A is at least $O(n^4)$".

- What does this mean? $f(n)$ is $\Omega(g(n))$.

- What does this mean? $f(n)$ is $\Theta(g(n))$.

- `Merge-Sort` is another sorting algorithm; it runs in $\Theta(n \log n)$ time.

  Consider the following sorting algorithm, called "Two-Face sort":

  ```
  Flip a fair coin
  **if** the coin lands on heads **then**
      **return** MergeSort($A_1 \ldots A_n$)
  **else**
      **return** BubbleSort($A_1 \ldots A_n$)
  ```

  Give O, Ω, and Θ bounds on Two-Face Sort.

- What is the running time of the following algorithm:

  `Linear Search`

  Input: A value x, an array of n values $A_1 \ldots A_n$

  Output: True if x is a value within A; False otherwise.

  ```
  $i = 1$
  **while** $i \leq n$ and $x \neq a_i$ **do**
      $i = i + 1$
  **if** $i \leq n$ **then**
      **return** True
  **else**
      **return** False
  ```

Which of the following algorithm running times has a better *growth rate*?

- $10 \log^{100} n$ or $\frac{1}{100}\sqrt{n}$?
- $100n^{100}$ or $1.01^{\frac{n}{100}}$?

- $\log^c n$ is $O(n^d)$, for any constants $c, d > 0$.
  In other words, a logarithm is the smallest non-constant polynomial.
- n^c is $O(d^{\frac{n}{e}})$, for any constants c and e, and any constant $d > 1$.
  In other words, a polynomial function grows slower than an exponential function in the limit.

Hierarchy of Running Times: Constant, Poly-logarithmic, Polynomial, Exponential

Rank the following functions from smallest to largest.

- $f_1 = n^n, f_2 = \log^2 n, f_3 = n^{1.0001}, f_4 = 1.0001^n, f_5 = 2^{\sqrt{\log n}}, f_6 = n \log^{1001} n$
- $f_1 = 2^{100n}, f_2 = 2^{n^2}, f_3 = 2^{n!}, f_4 = 2^{2^n}, f_5 = n^{\log n}, f_6 = n \log n \log\log n, f_7 = n^{\frac{3}{2}}, f_8 = n \log^{\frac{3}{2}} n, f_9 = n^{\frac{4}{3}} \log^2 n$

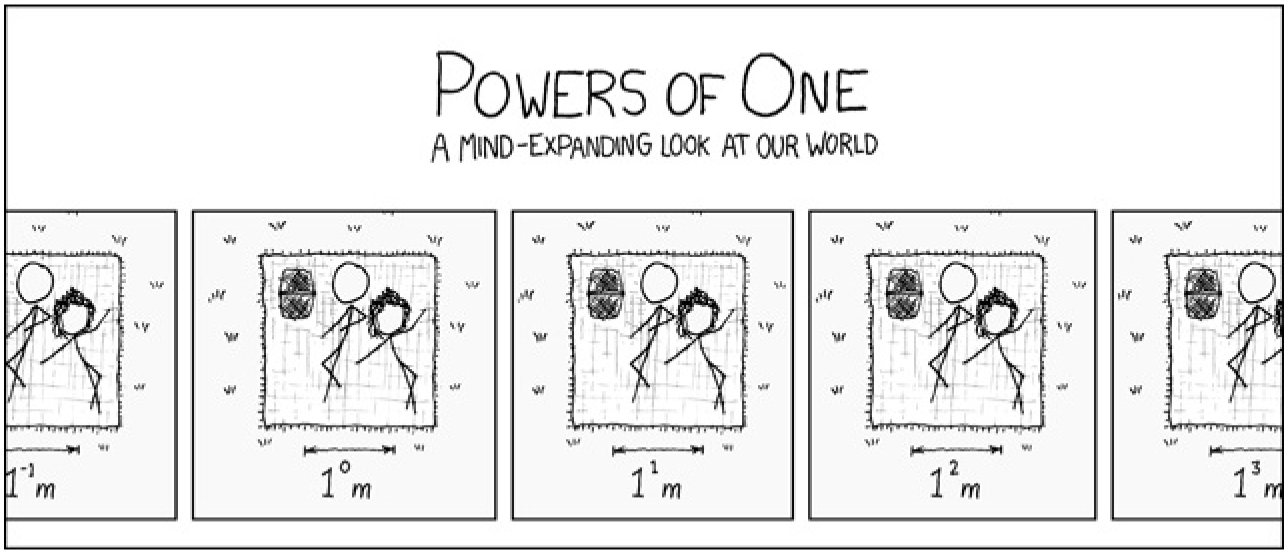

Figure 1: XKCD # 271. It's kinda Zen when you think about it, if you don't think too hard.

Assuming that a bit operation takes 10^{-11} seconds, here is a table of actual running times.

Problem Size	Bit Operations Used					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-7} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-5} s		*
10^3	1.0×10^{-10} s	10^{-8} s	10^{-5} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	10^{-4} s	10^{-3}s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-3} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-2} s	10.2 s	*	*

Entries marked with an * require so much time to complete that the heat death of the universe is expected to occur before these finish. I don't know about you, but I don't want to wait that long for an answer!

- Polynomial or better runtimes are considered **tractable**.
- Exponential or worse runtimes are considered **intractable**.

Whether or not a problem is *tractable* would be an important distinction. If it is, we can probably write an algorithm that will eventually give us our answer. If it is not, we might as well give up now.

There are some very important problems for whom we **do not know** whether they are tractable or not.

Properties of O-notation

- $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$. Prove or disprove: $f(n) + g(n) = O(h(n))$.
- $f(n)$ is $O(g(n))$. $\sum_{i=1}^{n} f(n) = ?$
- Suppose $g(n) = O(r(n))$. Prove or disprove: $f(n) - r(n)$ is $O(f(n) - g(n))$

Additional Exercises

How long would `Linear Search` take to run *on average*?